Integrating Profiling into MDE Compilers

Vincent Aranega, Antonio Wendell de Oliveira Rodrigues, Anne Etien, Frédéric Guyomarch, Jean-Luc Dekeyser

INTEGRATING PROFILING INTO MDE COMPILERS

Vincent Aranega[2], A. Wendell O. Rodrigues[1], Anne Etien[2], Fréderic Guyomarch[2], and Jean-Luc Dekeyser[2]

[1]PPGCC – Instituto Federal do Ceará, Fortaleza, Brazil
[2]INRIA Nord Europe, Lille, France

ABSTRACT

Scientific computation requires more and more performance in its algorithms. New massively parallel architectures suit well to these algorithms. They are known for offering high performance and power efficiency. Unfortunately, as parallel programming for these architectures requires a complex distribution of tasks and data, developers find difficult to implement their applications effectively. Although approaches based on source-to-source intends to provide a low learning curve for parallel programming and take advantage of architecture features to create optimized applications, programming remains difficult for neophytes. This work aims at improving performance by returning to the high-level models, specific execution data from a profiling tool enhanced by smart advices computed by an analysis engine. In order to keep the link between execution and model, the process is based on a traceability mechanism. Once the model is automatically annotated, it can be re-factored aiming better performances on the re-generated code. Hence, this work allows keeping coherence between model and code without forgetting to harness the power of parallel architectures. To illustrate and clarify key points of this approach, we provide an experimental example in GPUs context. The example uses a transformation chain from UML-MARTE models to OpenCL code.

KEYWORDS

Profiling – MDE – Traceability – Performance Analysis

1. INTRODUCTION

Advanced engineering and scientific communities have used parallel programming to solve their large-scale complex problems for a long time. Despite the high level knowledge of the developers belonging to these communities, they find hard to effectively implement their applications on parallel systems. Some intrinsic characteristics of parallel programming contribute to this difficulty, *e.g.*, race conditions, memory access bottleneck, granularity decision, scheduling policy or thread safety. In order to facilitate programming parallel applications, developers have specified several interesting programming approaches.

To increase the application development, software researchers have been creating abstraction layers that help themselves to program in terms of their design intent rather than the underlying architectures, (*e.g.*, CPU, memory, network devices). Approaches based on Model Driven Engineering (MDE), in particular MDE compilers, have frequently been used as a solution to implement these abstraction layers and thus accelerate system development.

MDE compilers take high-level models as input and a specific source code language is produced as output. Dealing with high-level models gives to the model designer a twofold advantages: on

DOI : 10.5121/ijsea.2014.5401

the one hand it increases re-usability and on the other hand it hides specific low-level details of code generation from the designed model. In this context, the system is directly generated from the high-level models.

However, the generated system can produce low performance issues due to a poor model design, even if optimization stages are proposed or implemented by the MDE compiler. In these cases, fine-tuning the generated code to improve the performances is a real need.

In an MDE approach, fixing the system implies applying modifications on one of the two artifacts: the designed models or the system source code. Directly modifying the generated source code is a difficult task for the model designer that does not know details about the target platform. Actually, the designer does not necessarily have the knowledge to efficiently modify the code. Moreover, the designed models lose the synchronization with the source code. A good solution to keep the coherence between the designed models and the generated source code is to regenerate code from the model correctly modified. Thus, changes aiming to achieve better performance must be made directly in the designed input models.

However, two issues make this method difficult to initiate as solution. Firstly, when a performance issue is observed, it is hard to figure out which parts of the models are responsible for this issue [15]. Thus, we have to keep a link between the models, the performance observations and the runtime results. Secondly, even if problems in the models are found, efficiently modifying it is not an easy task. Indeed, to provide better and more efficient changes in the models, the improvement must often take into account details of the target architecture, usually unknown and hidden to the designer.

Among the different techniques proposed to assist the designer during the performance improvement phase, two categories of tools can be found. The first one deals with static estimation computed in the model, whereas the second one, called profiling, deals with dynamic information. Our contribution focuses on profiling category because of its ability to gather details from a real execution environment. Hence, the recovered information comes directly from the system execution rather than from an estimation computed from the input high-level models.

In this paper, we present a framework to introduce performance optimization in MDE compilers. The main goal in this work is to provide a high-level profiling environment in a model design context where performance feedbacks are directly provided in the input models. With these visual feedbacks, model designers can easily identify the model parts producing poor performances. The performance information is recovered from dedicated profiling tools returning important measures as the running time or memories access time. This framework is based on model traceability to automatically return details and performance measures obtained during the system execution directly in the designed models.

In addition to the profiling information, the framework deals with dedicated advices, assisting the model designer quicly fine-tuning their models to achieve better results. These smart advices are computed by using the measured performances and specification details of the runtime plateform specification. Once the models modified according to the proposed approach, the system is then regenerated keeping coherence between the models and the code.

To validate our framework, we present in this paper two case studies in a MDE compiler towards GPU architecture. These cases studies take place in the *Gaspard2* [7] environment: an MDE framework for parallel-embedded systems. *Gaspard2* proposes, among others, an MDE compiler to several programming languages. The compiler takes UML models profiled with the MARTE

standard profile [13] as input model and generates system for few target platforms to reach simulation and validation purposes. Among the different target languages, *Gaspard2* includes an OpenCL branch introduced in [16][17]. Using such a framework, the designer can focus on the general system architecture without worrying about the Open Computing Language (OpenCL) [9] implementation details.

The paper is structured as follows: in section 2, we discuss about the major works of the domain. In section 3, we identify the different challenges we deal with in this paper. In section 4, we present how the profiling information is linked to the high-levels models. Then, in section 5, we illustrate this information feedback on an example in. In section 6, we present how the dedicated advices are computed before illustrating this advice computation on an example in section 7. Finally, we conclude and discuss about further works in section 8.

2. RELATED WORK

The analysis of software performance is part of the Software Performance Engineering (SPE) [19]. The SPE process uses multiple performance assessment tools depending on the state of the software and the amount of performance data available. SPE is a relatively mature approach and normally is associated to the prediction of the performance of software architectures during early design stages.

Several performance-modeling approaches have been proposed in the literature, including simulation-based or model-based approaches, often based on UML. Some examples of works, which have been developed in this research field, are enumerated below.

1. Model-Driven SPE (MDSPE) is proposed in [20] and deals with building annotated UML models for performance, which can be used for the performance predictions of software systems. MDSPE consists in deriving the performance models from the UML specifications, annotated according to the OMG profile for Schedulability, Performance, and Time (SPT) [12]. Many steps: *Performance Annotation* and *Performance Analysis* compose this approach. The first one deals with encapsulation of performance characteristics of the hardware infrastructure, as well as *Quality of Service* requirements of specific functions. The second one is implemented by a performance analyzer, which computes the performance metrics, which thereby predicts the software performance.
2. A simulation-based software performance modeling approach for software architectures specified with UML is proposed in [2]. Similarly to MDSPE, this approach is based on the OMG profile SPT [12]. Performance parameters are introduced in the specification model with an annotation. However, unlike the previous work, performance results estimated by the execution of the simula- tion model are eventually inserted into the original UML diagrams as tagged values, so providing a feedback to the software designer. The proposed method- ology has been implemented into a prototype tool called Software Architectures Performance Simulator (SAPS) [2].
3. The ArgoSPE approach, proposed in [8], is a tool for the performance evaluation of software systems in the first stages of the development process. From the designer's viewpoint, ArgoSPE is driven by a set of "performance queries" that they can execute to get the quantitative analysis of the modeled system. For ArgoSPE, the performance query is a procedure whereby the UML model is analyzed to automatically obtain a predefined performance index. The steps carried out in this procedure are hidden to the designer. Each performance query is related to a UML diagram where it is interpreted, but it is computed in a petri network model automatically obtained by ArgoSPE.

4. An interesting approach aiming at model refactoring is depicted in [10]. This approach relies on optimization of parallel and sequential tasks on FPGA accelerators. In this case, before generating the VHDL code, the application at Register Transfer Level (RTL) is analyzed by an optimization system, which exploits the allocation of FPGA resources. Then, the input model receives changes according to the optimization process results.

Further, there are several other research works that deal with SPE and UML but they are less co-related and we are not going into extended details. CB-SPE [3] reshapes UML performance profiles into component based principles. Model-Driven Performance Engineering (MDPE) [5] describes a model transformation chain that integrates multiparadigm decision support into multiple Process Modeling Tools. An extension [6] added to this work uses traceability and simulation engines in order to provide feedback to model designers. And in [15] is proposed a graph-grammar based method for transforming automatically a UML model annotated with performance information into a Layered Queueing Network (LQN) performance model [21].

As stated in the introduction, we base our work on the MARTE profile. It provides special concepts for performance analysis: Performance Analysis Modeling (PAM). This allows the model designer to define execution platform specification in the input models. However, as in UML SPT, the platform specification is modeled in the input model and assumes infrastructure knowledge from the model designer. Moreover, according to the specification, the performance analysis should be obtained from static estimations that may be very different from performances measured at runtime.

In short, all these earlier works lack the profiling feedback and possible directions aiming better performances in a real execution environment. Besides, they do not take into account small differences in features of the target platform and their relation with application behavior. Moreover, they impose to annotate the high level models, requiring a double expertise from the model designers. Indeed, they have to correctly design the models and correctly annotate them with runtime plateform specification. Our work does not require early annotations and relies on real execution environment aiming to make possible fine-tuning applications at design time.

3. PROFILING TOOLS INTO MDE COMPILERS: CHALLENGES

From the designed application point of view, these annotations are not required to generate the software. Moreover the runtime plateform specifications are a simple knowledge attached to the hardware, which could be managed out of the designed model. This paper addresses the challenge to take benefit from information about performance of the generated software provided by an existing profiling tool.

The general process is sketched in Figure 1a. It is made of three main parts (noted *A*, *B* and *C*) that represent a classic profiling integration in a software life cycle: **part A** represents the software compilation, **part B** corresponds to the software execution, and **part C** sketches the profiling information inclusion.

For an MDE approach, the flow is detailed as follow. Once we have the high-level models, the MDE compiler generates the software (step 1, named *transformations chain* in Figure 1a). The software is then compiled and executed (step 2 and 3). Thanks to a profiling tool, several performance measures are gathered (step 4) and are brought back in the initial high-level models (step 5). Although our methodology does not impose a rigid workflow, our approach relies on two major activities: first we run the code exactly as it is generated from the original input model,

then the application designer analyzes the runtime behavior based on profiling feedback annotated on the input model; second, the designer, taking into account the provided information, modifies the model aiming to obtain better results. Once the model is modified, the code is again generated and then executed in order to verify the result of theses changes. This simple flow addresses many challenges linked to the transformation from abstraction layers to others.

Indeed, the obtained profiling information gives execution details, which are only linked to the generated source code lines (e.g, a time measure is relative to a function or a variable in the source code). This profiling information must be linked to input models to be really useful for the model designer. How to propagate profiling information to the design model is, therefore, an issue, which must be resolved. As first answers to this issue, the model transformation traceability is interesting because it allows keeping links between the models during transformations steps.

Dealing with the profiling information raises another issue. The software performances gathered by the profiling tool give profiling information about the source code and could be far from the designed model (e.g, about memory consumption or time measure which can be comprehensible only on source code). They might be useless and hard to analyze to adequately modify the designed model. In this context, handling the runtime plateform details, as memory size or the number of thread allowed can be useful. Indeed, they could be analysed with the profiling information in order to give a clue on the way to modify the software. To ease the analysis, the profiling information conjointly used with an automatic analysis could give more adequate information to the model designer.

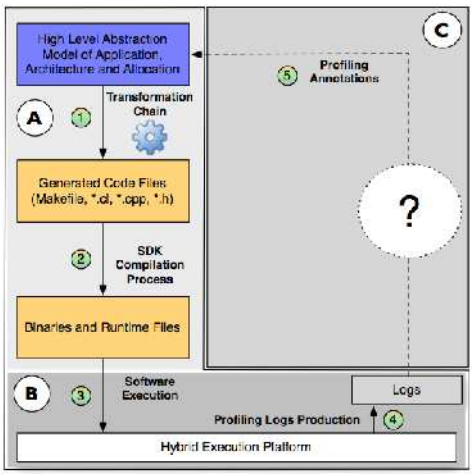

Figure 1a. Towards Integrating Profiling and MDE Compilers

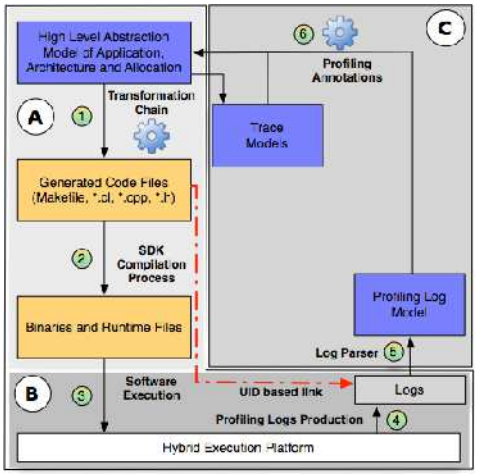

Figure 1b. Approach Overview

To propose a profiling solution for MDE compilers the two following main issues must be solved:

- how to connect the profiling information to the high level models
- how to bring the profiling information intelligible for the high level models

In order to introduce solution for these issues, we propose in this paper an adaptable framework helping MDE compiler developer to add a profiling capability to their compilers. Beyond the issues we raised in this section, the framework relies on the following statements regarding the MDE compiler (*transformation chain*):

- the transformation chain must have a model transformation traceability support
- a profiling tool must exist for the target language

On the one hand, the model traceability support is a first answer to the "how to connect" issue. For a given transformation chain (MDE compiler), it helps to link elements from the input of the chain to their respective generated elements at the end of the chain and *vice versa* (we discuss about the model transformation traceability in section 4.3). On the other hand, using an existing profiling tool allows to take benefits from expert development and it supposes that the obtained profiling results are trustworthy.

4. FROM EXECUTIONS TO MODELS

In this section, we address the first challenge we previously discuss:

- how to connect the profiling tool information to the high level models

Our approach for the profiling information feedback is sketched in Figure 1b. The **part A** (code generation) is similar to the one presented in Figure 1a except that traceability is introduced. The presented profiling life cycle follows a classic structure:

1. the software is generated from the high-level models (step 1) and the trace models are produced
2. the software is executed, producing profiling logs (steps 2 to 4)
3. the produced logs are returned in the input models (steps 5 to 6).

Currently, in works available through the literature, only the first part of the process is automatic (step 1). The second part (steps 2 to 4) producing the logs highly depends on the used tools. The third part where logs are analysed and connected to the input elements that should be modified remains a manual and complex process. In this paper, we focus on this step of the process (steps 5 to 6 in Figure 1b) by automating it.

In order to automate the profiling information feedback, our process uses: the model-to-model traceability (*Trace Model*).

In this section, we present how the traceability is managed in the compilation chain and the required modifications on the model compilation chain. Then, we present how the profiling information are reported into the high-level models.

4.1. Managing The Whole Chain Traceability and Avoiding Model-to-Text Traceability

In order to keep the links between the input models and the software execution, trace models are produced all along the compilation chain, except for the model-to-text transformation. The translation from model to text implies keeping information on text blocks and words [14]. The granularity for this kind of trace made its management and maintainability difficult. In our case, the code has to be studied only in term of the abstract concepts from the models, and not in terms of blocks and words. To be coherent with this idea, in this paper, the model-to-text traceability has been avoided.

To bypass the model-to-text trace, the code generation deals with unique identifiers (UIDs) associated to each element in the last model of the transformation chain. The profiling logs produced by the software execution refer to the UID of the analysed element. Thus, the Profiling

Logs can be rebound to the model elements. Concretely, in order to generate the UIDs, we use the Eclipse Modelling Framework (EMF) feature called *Universal Unique IDentifier* (UUID) and, consequently, we modify the compilation chain. A new transformation only adding the UID is inserted as last step of the model-to-model transformation chain, just before the code generation.

4.2. Profiling Logs Parsing

According to the used Software Development Kit (SDK) and profiling tools, these profiling logs are generated with a dedicated format. This format is parsed using a shell-script that builds a profiling model conform to the metamodel presented in Figure 2. This metamodel is generic enough to produce models gather information that can be found in the profiling logs.

The metamodel root: *ProfilingModel* gathers several *LogEntries*, which represent the profiling entries from the logs. In addition, *ProfilingModel* specifies the running hardware model (*e.g.* Tesla T10 or G80 in GPU context) with the *archiModel* at tribute. Each *LogEntry* contains several *Parameter* elements owning a *kind* and a *data* representing: the information type (*e.g.*, occupancy, time or memory consumption), and its value. In order to keep the link between the profiling information and the transformation chain, each *LogEntry* keeps the introduced UID from the logs. In addition, a *timeStamp* attribute is added to the *LogEntry* in order to keep the logs sequence.

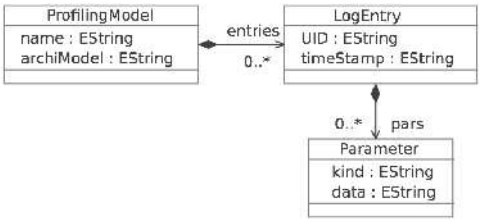

Figure 2. Profiling metamodel

4.3. Backtracking Profiling Information in the Input Model

We have argued in section 3 that a link must be kept from the profiling logs to the designed model. With the UIDs use, we full in the gap between execution logs and models. Now, in order to report the profiling information back in the designed models, we must exploit the model transformations traceability links.

Model transformation traceability keeps links between the elements consumed and produced during a transformation execution. All of these links, produced for one transformation, are gathered in a trace model. However, MDE compilers usually deal with transformations chains implying multiple transformation and thus multiple trace models. To maintain the traceability links on a whole transformation chain, we deal here with our own traceability mechanism [1], based on two specific trace metamodels.

The first one, the local trace metamodel, is used to keep the traceability link for a single transformation, whereas the second one, the global trace metamodel, is used to maintain the local trace order in a transformation chain. These two metamodels enable the identification of the high-level model elements that have lead to the creation of elements in the model used for the code generation.

The profiling information feedback is performed in two steps. First, each *LogEntry* (Figure 2) is linked to the elements it refers to in the model used for the code generation. Indeed, the UIDs contained in the *LogEntries* refer to elements in the last model before the code generation. Then, once the element referred by the UIDs are found,

the model transformation traces are backward navigated in order to recover the input elements producing the profiling information. During the backward navigation, two cases can occur, the retrieved elements are reduced to one or several elements.

If only one element is found, the advice is simply reported on it. However, if multiple elements are found, a different strategy must be applied because reporting the advice on all retrieved elements can create confusion. To solve this issue, the expert system can be configured to specify some element types of the input metamodel for each profiling information. From this way, only the input elements of the specified types are kept. Finally, the profiling information is connected to these elements in the input models. In the following section, we illustrate our report mechanism on a case study.

5. EXAMPLE AND BENCHMARKS

5.1. UML/MARTE to OpenCL Chain

The *Gaspard2* environement [7] is used for SoC co-design and contains many MDE compiler chains. Among them, a branch of *Gaspard2* allows to generate OpenCL [9] code from an application model specified in UML-MARTE. It aims to provide resources for application development to non-specialists in parallel programming. Thus, physicists, for instance, can develop applications with performances comparable to those manually developed by experienced programmers. Table 1 presents the set of model transformations that compose the chain towards OpenCL. The whole process is detailed in [18].

Table 1. OpenCL Transformation Chain

#Transformation	Description
1: UML to MARTE Metamodel	This transformation adapts a model conforming to UML to a model conforming to the MARTE metamodel. Thus, remaining transformations do not need to deal with all unnecessary extra complexity of UML.
2: Instances Identification	This transformation adds instances of ports for each part within a component. In order to easily identify local variable.
3: Tiler Processing	This module transforms every tiler connector [4] to tiler tasks.
4: Local Graph Generation 5: Global Graph Generation 6: Static Scheduling Policy	These transformations are responsible for the definition of task graphs and the application of a simple scheduling policy.
7: Memory Allocation	It regards to memory handling, variable definition, and data communication.
8: Hybrid Conception	It summarizes all explicitly modeled or implicitly defined elements by earlier transformations into a single structure.
9: Code Generation	This model-to-text transformation transforms the earlier analyzed elements directly to source code according to the OpenCL syntax.

5.2. UML/MARTE to OpenCL: Prerequisites and Adaptation

Some adaptations are required before applying our apporach in order to satisfy the prerequisites. As shown in the previous section, to introduce our approach in an existing transformations chain, some prerequisites and adjustments are necessary. The traceability mechanism is already

supported in *Gaspard2*; the OpenCL transformation chain has to be modified to manage UIDs and a useful profiling tool must be used.

5.2.1. Profiling Tool for OpenCL

For OpenCL code profiling, we use a tool proposed by NVidia: ComputeVisualProfiler [11]. This tool performes several measures during the application execution on the GPU and reports them in a classical text file (i.e, profiling log). This profiling tool uses the function names as pointer for the developers to help them to figure out on which part of the code the profiling information are relative to. From this profiling log, the profiling tool can produce a graphical view of the obtained profiling information. The generated logs also contain information about the execution platform, such as the GPU model used.

5.2.2. Adding UIDs in the Transformation Chain

A model transformation is added to the OpenCL transformation chain to introduce the UIDs in the generated source code. This transformation is inserted before the code generation (the 9th transformation in Table 1). As the profiling tool uses the function names as pointers and as they are generated from the element *name* attribute, the added transformation concatenates, for each element, the element UID to the element *name* attribute. By this way, the elements UIDs are present in the profiling logs when the generated software is executed.

5.3. Case: Conjugate Gradient Solver

The case study we propose in this paper is a complete example which presents an application design, code generation and the profiling feedback. The example is based on the Conjugate Gradient (CG) algorithm.

The CG is a common numerical algoritm used to solve large linear systems. Here, this method is the masterpiece of a generated application, part of an industrial simulation tool and its input data comes from a mesh representing an electrical machine: in our example the machine is an automotive alternator developed by VALEO[1]. Alternators are AC electrical generators and usually the word refers to small rotating machines driven by automotive.

We have applied the approach on this real industrial example. And this produces a linear system whose matrix A has n = 775,689 and nnz = 12,502,443. Benchmarks for this example gives speedup of 9x with relation to standard Fortran version on CPU. We had 10,000 iterations in about 2300 seconds on CPU against 250 seconds on GPU. This is a good result that shows the potential increase in speedup according to complexity of the problem. From this application, two operations are analyzed in order to apply our approach. The *dot product* that is depicted in Subsection 5.4, and the *DAXPY* that is used as a case study in Section 7.

For all the tests, the configuration environment was as follows:

- CPU AMD Opteron 8-core @2.4GHz and 64GB RAM;
- GPU NVidia S1070 4 devices Tesla T10 (4GB RAM each) - Compute Capability 1.3;
- Linux, GCC 4.1.2, OpenCL 1.0.

5.4. Task Distribution of the Dot Product

[1] www.valeo.fr

Dot product operations consist in two parts: a scalar multiplication, element by element and then the sum of all these products. This last step is called a reduction and can be performed partially in parallel, but it always come to a point where the last addition is performed sequentially. This very last step is usually done on the CPU. Then a model designer must decide when to remove the last reduction from GPU (see Figure 3). In fact, once the synchronization of work-items in GPUs is made at work-group level, the final reduction *r2* must be either ended on the CPU, either on the GPU but this last solution needs to relaunch a new kernel. Because of this overload, the final parallel reduction runs on the host side. For instance, in Figure 3, we perform the dot product on vectors of 2516892 elements. The host launches 492 work-groups containing 512 work-items. Each work-item performs the product between two corresponding elements. A special elementary task (*r:Reduc*) is deployed by a special IP that does a parallel reduction and selects only the first iteration to write out the reduction result. At the end, 492 results are produced. The best "candidate" solution is to perform the final reduction on the host side (CPU). In such a scenario, it is necessary to take into account the size of vectors. Indeed, for huge vectors the number of work-items per work-group can exceed the hardware limits. We do not provide any automatic proposal for this situation. However, the profiling feedback can help the designer to identify either this is really the best solution or, otherwise, to allocate the final reduction onto the GPU. Information about execution times can provide the answer.

5.4.1. From Logs to Annotated Model

Our approach allows annotating directly on the high-level model information about runtime operations. The proposed methodology extracts profiling results, transforms them into model elements, and, by using the UID (discussed in 5.2.2), it is able to report them as a comment onto the *allocate* element. This information allows the model designer to take a decision for the more suitable allocation choice.

```
1  # OPENCL_PROFILE_LOG_VERSION 2.0
2  # OPENCL_DEVICE 0 Tesla T10 Processor
3  # OPENCL_PROFILE_CSV 1
4  # TIMESTAMPFACTOR ffff6c654a49c58
5  timestamp,gpustarttimestamp,method,gputime,cputime,ndrangesizeX,ndrangesizeY,
       workgroupsizeX,workgroupsizeY,workgroupsizeZ,stapmemperworkgroup,regperworkitem,
       occupancy,streamid,local_load,local_store,gld_request,gst_request,memtransfersize,
       memtransferdir,memtransferhostmemtype
6  224099.000,127b3c82a4ec62a0,memcpyHtoDasync,182.016,889.000,,,,,,,,1,,,,,1006756,1,0
7  225202.000,127b3c82a4f7f2c0,memcpyHtoDasync,195.040,556.000,,,,,,,,1,,,,,1006756,1,0
8  225819.000,127b3c82a4fdb920,PPdotProd_KRN__sb9LkTquEeGg4rUXqAwjQA,143.648,
   263.000,492,1,512,1,1,36,6,1.000,1,0,0,544,289
9  226184.000,127b3c82a502f6a0,memcpyDtoHasync,4.928,89.000,,,,,,,,1,,,,1968,2,0
```

Figure 3. Profiling Results Extract in CSV Format

The profiling environment creates a log file in CSV format having some dynamic measured data (as seen in Listing 1). The file header (line 1 to 4) contains data about the target platform. The code run in a Tesla T10 GPU in this case (line 2). The remaining list consists in description of fields and log entries. The description of fields (line 5) indicates in which order they will appear in a log entry. For instance, in Listing 1, a log entry begins with the *timestamp* field. The second field that can be retrieved in an entry is *gpustarttimestamp*, then *method*, and so on. For each entry, the *method* field indicates the GPU *kernel* call (line 8), except for the ones with the *method* field set to *memcpyHtoDasync* and *memcpyDtoHasync* (corresponding to memory copies, lines 6, 7 and 9).

[2] This number comes from industrial simulations.

This profiling file is analyzed by a shell-script parser, which converts it to XMI format that conforms to the profiling metamodel depicted in Figure 2. The model (Figure 4) created from the CSV log file gathers exactly the same information. Except the *timestamp* field and the UID contained in the *method* field that becomes attributes of the *LogEntry*, all the other fields (*e.g. gputime, cputime or ndrangesizex*) are transformed into *Parameters* with the value of the log entry. Listing 1 (highlighted elements) and Figure 4 present the UID parameter with value *_sb9LkTquEeGg4rUXqAwjQA* for the kernel call with timestamp equals 225819.000.

The profiling information feedback algorithm uses the UID retrieved from the profiling logs. From the Listing 1 and Figure 4, the highlighted UID corresponds to an element in the model produced by the "add UID" model transformation; the element with this UID is found in the model before the code generation.

Table 2 – Element Set Gathered by the Backward Navigation

Element found	High-level Elements
PPdotProd_KRN : Kernel	*Multiply : Class* *m : Property* *frommtogp : Abstraction*

Table 2 gathers the found element[3] and the elements retrieved from the backward trace navigation from it. The profiling information is relative to the *PPdotProd_KRN*: *Kernel* element, which has been generated from the high-level models by 3 elements. The profiling information must thus be linked to the 3 retrieved elements. However, we decided to handle the time relative information differently from the others. Instead of attaching the information to each retrieved elements, we decided to attach the information only to *Abstraction* type elements. Indeed, time relative measures strongly depends on the way the tasks are placed on the hardware, thus the *Abstraction* link is the better candidate to report the information on. So, in our example, the time relative information is attached to the *frommtogp*: *Abstraction* element (*Abstraction* stereotyped with *allocate* in Figure 5). The other information coming from the profiling log has been removed for reading purposes.

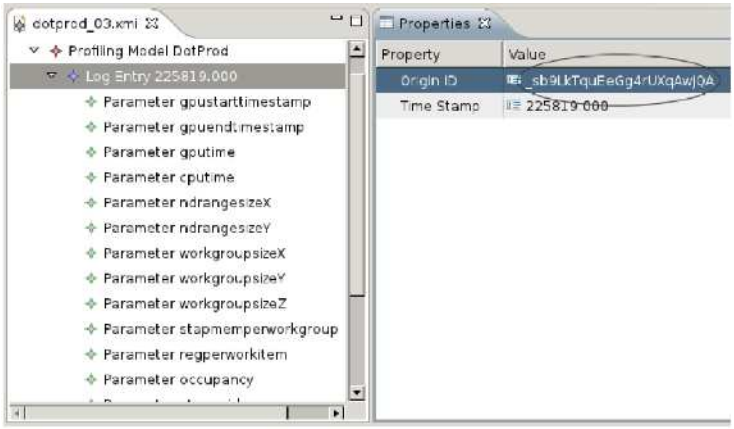

Figure 4. Profiling Results Model

[3] In this table, we have removed the elements UIDs for readability reasons.

5.4.1. From Logs to Annotated Model

Figure 6 shows us the valuable difference with respect to the execution time on the CPU. The operation takes 3.5ms on the CPU while it takes 0.72ms on the GPU. Such a result makes the model designer deciding to switch the allocation of the last reduction task from the CPU to the GPU. Despite its visual representation, this indication is even more relevant when the model designer has runtime details and can change only a single arrow to choose another processor to run the operation. All the complex lines of OpenCL code about data transfers, kernel launches, and so on, are avoided. Thus, the new generated code is ready, useful, and achieves more performance than the initial one.

In the previous sections, we show how information that directly came from the profiling tool are brought to the high-level models. This information feedback from the software execution defeats the first challenge exposed in section 3. In this case, the added comment is comprehensible in the inputmodel, but it is not always the case.

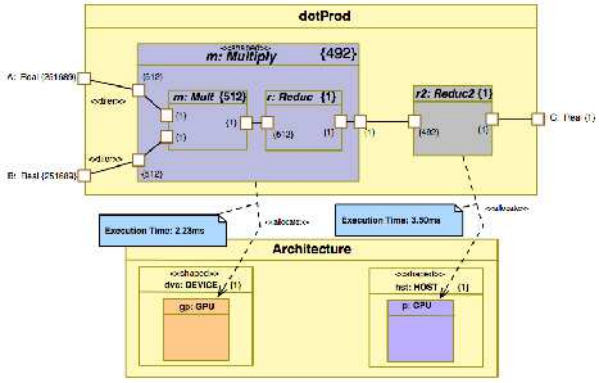

Figure 5. Summary of the UML-MARTE Model of Dot Product - Final Reduction on CPU

In the previous sections, we show how information that directly came from the profiling tool are brought to the high-level models. This information feedback from the software execution defeats the first challenge exposed in section 3. In this case, the added comment is comprehensible in the input model, but it is not always the case.

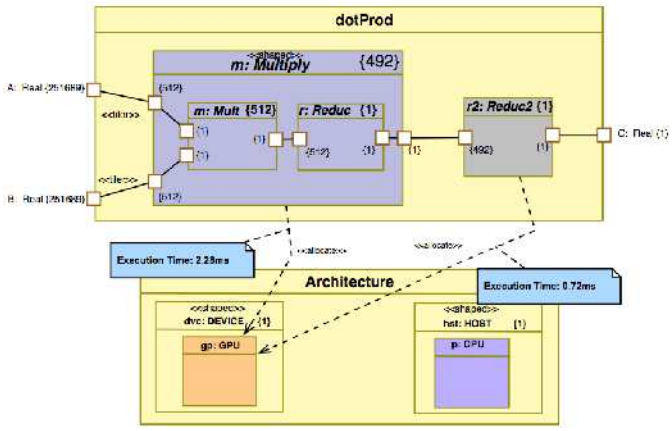

Figure 6. Summary of the UML-MARTE Model of Dot Product - Final Reduction on GPU

6. HELPFUL ADVICES FOR MODEL DESIGNERS

In this section, we address the second challenge we previously discussed:

- how to bring the profiling information comprehensible for the high level models

Indeed, as we explained in section 3, the profiling information can be close to the generated code and, therefore, some profiling information is potentially not useful or hard to interpret for the models designers. Thus, the profiling information should be analysed to give a more intelligible execution feedback. Figure 7 illustrates our approach with the profiling log analysis. Only the **part C** changes, by integrating an expert system, represented by the α-box. The expert system contains a knowledge base about the runtime devices (named *Device Features Database Model*) and a set of formulas (named *Domain Specific Profiling Analysis Transformation Library*), currently represented by model transformations. The different formulas represent the expert system heart and take into account two information sources, the profiling tool and the knowledge base.

In this section, we present the expert system, handling both information sources to produce high-level information intended for the models designers.

6.1. An Expert System to Produce Advices

More than profiling results, we are able to provide smart advices to the model designers. The expert system uses the profiling logs that give factual data about execution whereas the hardware knowledge base gives features about the runtime platform. By combining both sources with dedicated analysis, it is possible to deduce how to improve the execution of the generated code. For instance, assuming the device supports 32MB in shared memory allocation per thread group and the application allocates at runtime 48MB. The expert system is able to indicate that it is necessary to decrease the memory allocation after analysing profiling log results and device constraints. In this case, the expert system provides a hint, in the designed model, where the problem occurs. In order to analyse the many properties of the results, an extensible library (*cf.* Figure 7) is proposed in this paper.

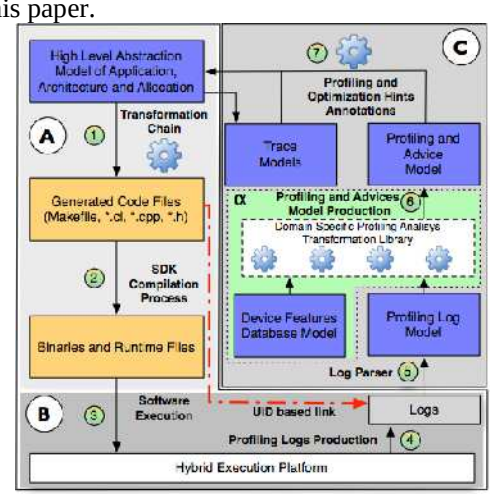

Figure 7. Approach Overview

6.2. Smart Advice Computation

The expert system includes a number of algorithms that can estimate the best values to specify in the high level model in order to improve the modelled software performances. These algorithms are implemented by model transformations. We use model transformations for the simplicity and convenience of handling input models they provide (obviously, any programming language with the ability to navigate and produce models can be used for writing the algorithms). The results produced by the algorithm are presented in an advice model.

6.3. Backtracking Advices in the Input Models

As for each *LogEntry* of the profiling model, the advices computed by the expert system are reported in the high-level models. As each *Advice* owns a UID attribute, inherited from the profiling model, the advice feedback is performed as for the *LogEntry*. The *Advice body* attribute contains the computed advice, as well as the profiling information. Thus, only one backward navigation algorithm is used.

7. CASE STUDY: DAXPY MODEL

In this section, we show how the GPU knowledge base is build and how the advice proposed by our expert system can help to modify high-level models and improve GPU performances for the Conjugate Gradient algorithm.

7.1. Creating the GPU knowledge base

In the OpenCL transformation chain context, the hardware knowledge base must be built for GPUs. The metamodel we obtain and we consider in this section is the one illustrated in Figure 8. It is obvious that the device features models produced from this metamodel, GPUs oriented, must be produced by GPU experts.

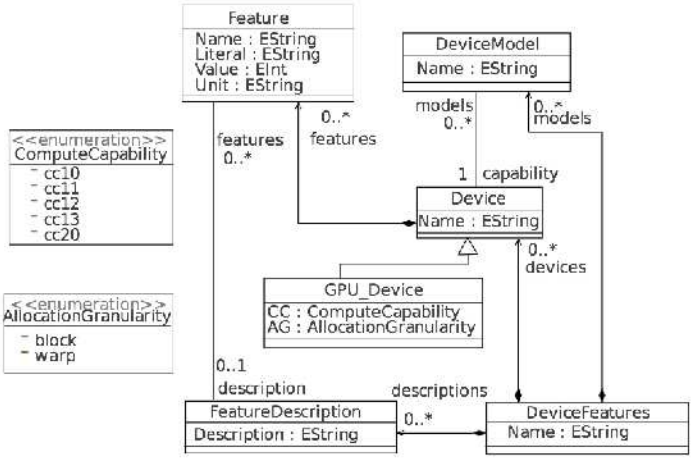

Figure 8. GPUs Device Features Metamodel

In Figure 8 we have two enumerations: ComputeCapability and AllocationGranularity. These two enumerations are used by the attributes CC and AG from the GPU_Device metaclass. The first attribute represents the GPU computing capacity (e.g, cc10 represents the computing capacity

version 1.0). The second attribute represents the granularity of the allocations performed by the GPU.

As an example, Figure 9 shows an excerpt of the device feature model (hardware knowledge base) for *GPU devices*. In this excerpt, 10 NVidia' (here *a.k.a.* GPU 1.3) features are shown. The highlighted feature: *TW*, gives a number of 32 threads per *Warp* (or work-items in OpenCL terminology). This information represents the quantity of threads that can be active at the same time on the hardware. Although this metamodel was designed according to vendor's features, it can be modified or extended to comply with other vendor device models.

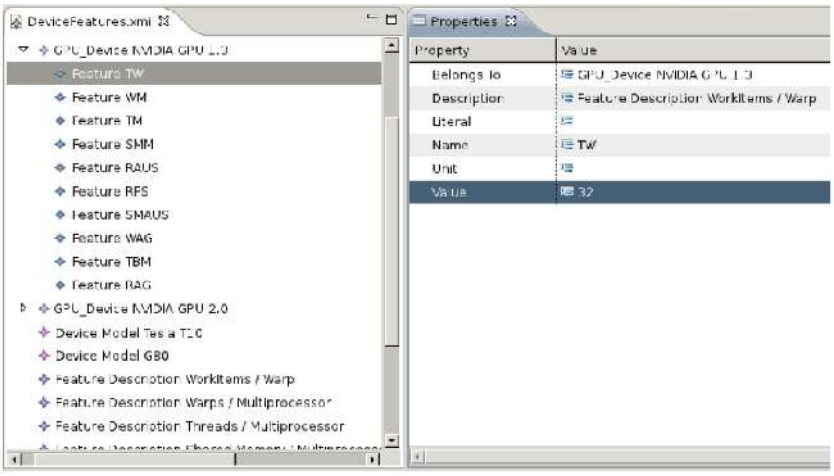

Figure 9. GPU Device Features Database Model

7.2. Analyzing Processor Occupancy on BLAS DAXPY Operations

DAXPY is essentially an *O(N)* parallel operation. It is a linear combination of vectors $y = \alpha x+y$. The UML-MARTE model for the *DAXPY* present in the CG application is shown in Figure 10b. The application's vectors are arrays of 775,689 elements. This number results from the model of an alternator from VALEO and takes advantage of the massively parallel processors provided by GPUs. This operation as the previous one is part of simulation tool that had its performance improved, resulting in a global speedup of the order of 9x. This result is particularly valuable to VALEO for larger simulations that require long time to finish. The composed component *DAXPY* instantiated in the program consists of a repetitive task *xy:DaxpyGroup*. In our application, this kind of task is composed by operations on single elements. Repetitive tasks are potentially parallel and are allocated onto GPU.

The repetition shape of the task in this case is *{16,48494}*, i.e. the task operation runs 775,696 (the product of the shape) millions times on one element of each vector whose size equals 775,689. The total of work-items is calculated by multiplying the dimensions of the task hierarchy. The first one becomes the number of work-items and the second one the number of groups. The definition of this shape is a decision of the designers and usually they take into account the *Intellectual Property (IP)* interface associated to the elementary task and its external *tilers* [4]. Moreover, considering that, in "compute capability" 1.x GPU devices, memory transfers and instruction dispatch occur at the Half-Warp (16 work-items) granularity, it is reasonable to define groups composed of work-items at a first try.

Once the application is designed with all necessary elements, we generate all source code files. In addition, trace models are generated for each model-to-model transformation thanks to the traceability mechanism.

7.3. Profiling Feedback

Among all the measures coming from the profiler, the kernel occupancy factor has an important impact on performance. Usually the aim at executing a kernel is to keep the multiprocessors and, consequently, the devices as busy as possible. The work-items instructions are executed sequentially in OpenCL, and, as a result, executing other warps when one warp is paused or stalled is the only way to hide latencies and keep the hardware busy. Some metric related to the number of active warps on a multiprocessor is therefore important in determining how effectively the hardware is kept busy. This metric is called occupancy. The occupancy is the ratio of the number of active warps per multiprocessor (WPM) to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that are actively in use. Hence, higher occupancy does not always equate to higher performance, there is a point above where additional occupancy does not improve performance. However, low occupancy always interferes with the ability to hide memory latency, resulting in performance degradation.

The important features to compute the occupancy vary on the different GPU "compute capability" are:

- the number of registers available;
- the maximum number of simultaneous work-items resident on each multiprocessor;
- and the register allocation granularity.

The number of work-items resident on a multiprocessor relies on index space as known as *N-Dimensional Range (NDRange)*. The OpenCL chain computes the information from the shape of the task, which will become a kernel. Hence, changes in the dimensions of shape affect the *occupancy*. From the point of view of the proposed approach, *occupancy* is a specialized module that can be included to the expert system. For other analysis other specialized module can be added to attain specific goals. For this example, we analyze the occupancy of the multiprocessors. Occupancy is function of constant parameters (features) from device and some measures directly obtained from the profiler.

The process of calculating occupancy is implemented in a QVT transformation. This transformation takes two input models (according to Figure 7): the *Device Features Database* and the *Profiling Logs*. In this example the first one conforms to a metamodel based on NVidia GPUs. For instance, from the model presented in Figure 10a we see that the GPU Tesla T10 has computed capability equals 1.3 and its warps contain 32 threads (or work-items in OpenCL terminology).

7.4. Benchmarking

For this example the first running gives the results illustrated in Figure 10a. The application launches 48,481 groups of 16 work-items onto the device. Figure 10a shows that we got only 25% (0.250) of the multiprocessor occupancy. Our goal is to increase the occupancy and decrease the relative GPU execution time.

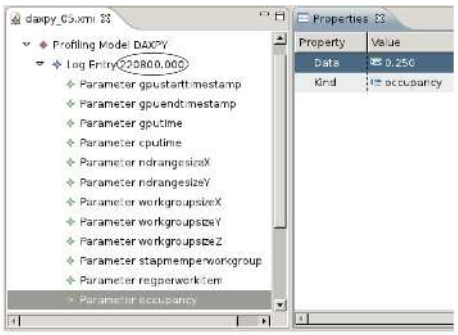

Figure 10a. GPU Device Features Database Model

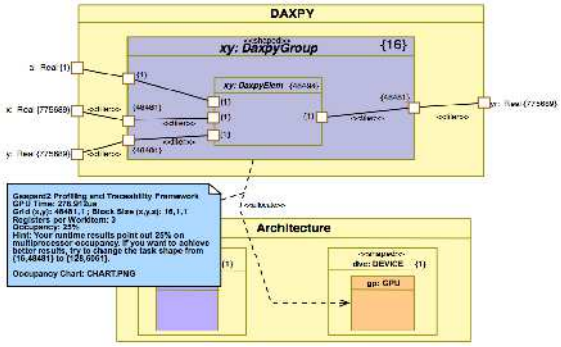

Figure 10b. GPU Device Features Database Model

By using our approach results are combined with GPU features and this returns a smart advice as comment in the input UML-MARTE model (Figure 10b). Besides the performance parameters available directly on the comment, a *hint* points out a possible change in the model to improve the generated code. Additionally, the advice provides an image reference of a chart (as seen in Figure 11) for all predicted occupancy according to these results. In this case it is suggested to change the task shape from {16, 48481} to {128, 6061}.

A simple analysis search for the first block size gives us 100% occupancy. For instance, the expert system automatically highlights the first (block size=128) and second (block size=256) maximum values in the chart.

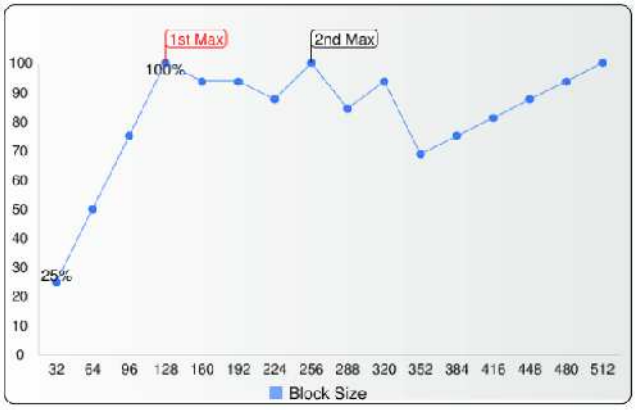

Figure 11 – Occupancy by Varying Block Size

After executing the modified application, we have 100% on occupancy and a reduction of the GPU time as it is shown in the new profiling log (Listing 3, line 8). Previously we had 278.912µs, and then the GPU time decreased to 82.048µs. As expected, the modified model achieves better performance than the original one. The whole execution of the kernel is about 2.3x faster.

8. CONCLUSION AND FUTURE DIRECTIONS

In this paper we address performance improvements as part of the life cycle of the application design and execution. We provide a high level profiling environment for MDE compiler that we applied for OpenCL in the context of the *Gaspard2* environment. This environment allows the

model designers to efficiently modify their models to achieve better performances. The profiling environment is based on two main artifacts: an expert system and a traceability mechanism.

The expert system proposed here uses data from a feature database dedicated to the runtime platform and profiling logs. The aim is to compute a smart advice explaining how the model should be modified in order to achieve better performances. From the application designers point of view, they do not necessarily need to know complex details about the runtime platform. Moreover, no performance specification is given in advance. The expert system summarizes the profiling logs and minimizes the tasks of the model designer by analyzing the gathered profiling data.

Especially for GPU applications, better performances rely on speedup, memory use and processor occupancy. In order to provide this feedback, this paper proposed to retain coherence between code generation and traceability. Thus, the expert system uses the traceability to return a UML comment consisting of a computed smart advice and profiling logs on the specific input model elements that involve the analyzed performance issue.

Obviously, although our case study is focused on GPU applications, our approach can be adapted to other environments with code generation and profiling tools. Indeed, as we have shown in the paper, the library that is part of the expert system can be extended to analyze other issues or other devices. Moreover, the traceability mechanism that we provide can fit to any transformation

[3] Antonia Bertolino and Raffaela Mirandola. Towards Component- Based Software Performance Engineering. In Component-Based Software Engineering, 2003.

[4] Pierre Boulet. Array-OL Revisited, Multidimensional Intensive Signal Processing Specification. Technical report, 2007. Available from: http://hal.inria.fr/inria-00128840/PDF/RR-6113v2.pdf.

[5] Mathias Fritzsche and Wasif Gilani. Model transformation chains and model management for end-to-end performance decision support. In Proceedings of the 3rd international summer school conference on Generative and transformational techniques in software engineering III, GTTSE'09, pages 345–363, Berlin, Heidelberg, 2011. Springer-Verlag. Available from: http://portal.acm.org/citation.cfm?id= 1949925.1949936.

[6] Mathias Fritzsche, Jendrik Johannes, Steen Zschaler, Anatoly Zherebtsov, and Alexander Terekhov. Application of Tracing Techniques in Model-Driven Performance Engineering. In In Proccedings of the 4th ECMDA-Traceability Workshop (ECMDA'08, June 2008.

[7] A. Gamatié, S. Le Beux, E. Piel, R. Ben Atitallah, A. Etien, P. Marquet, and J-L. Dekeyser. A Model Driven Design Framework for Massively Parallel Embedded Systems. ACM Transactions on Embedded Computing Systems (TECS)., 10(4), 2011.

[8] E. Gómez-Martínez and J. Merseguer. ArgoSPE: Model-based Software Performance Engineering. volume 4024, pages 401–410. Springer-Verlag, Springer-Verlag, 2006.

[9] Khronos Group. OpenCL - The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/. Available from: http://www.khronos.org/opencl/.

[10] Sébastien Le Beux. Un Flot de Conception pour Applications de Traitement du Signal Systématique Implémentées sur FPGA à Base d'Ingénierie Dirigée par les Modèles. These, Université des Sciences et Technologie de Lille, December 2007.

[11] NVIDIA. Compute Visual Profiler. http://developer. download.nvidia.com/compute/cuda/4_0_rc2/toolkitMdocs/ VisualProfiler/Compute_Visual_Profiler_User_Guide.pdf, 2010.

[12] OMG. UML Profile for Schedulability, Performance, and Time, version 1.1. http://www.omg.org/spec/SPTP, 2005. Available from: http://www.omg.org/spec/SPTP.

[13] OMG. Modeling and Analysis of Real-time and Embedded systems (MARTE), Version 1.0. http://www.omg.org/spec/MARTE/1.0/, 2009. Available from: http://www.omg.org/spec/MARTE/1.0/.

[14] Gøran Olsen and Jon Oldevik. Scenarios of Traceability in Model to Text Transformations. In Model Driven Architecture- Foundations and Applications, Lecture Notes in Computer Science. 2007.

[15]Dorina C. Petriu and Hui Shen. Applying the uml performance profile: Graph grammar-based derivation of lqn models from uml specifications. In Proceedings of the 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools, TOOLS '02, pages 159–177, London, UK, 2002. Springer-Verlag. Available from: http://portal.acm.org/citation.cfm?id=647810. 737982.

[16] A. Wendell O. Rodrigues, Frédéric Guyomarch, and Jean-Luc Dekeyser. An MDE Approach for Automatic Code Generation from MARTE to OpenCL. Technical report, INRIA Lille - RR-7525, 2011. http://hal.inria.fr/inria-00563411/PDF/RR-7525.pdf/.

[17] A. Wendell O. Rodrigues, Frédéric Guyomarch, and Jean-Luc Dekeyser. An MDE Approach for Automatic Code Generation from UML/MARTE to OpenCL. IEEE Computer in Science & Engineering - Special Edition on GPUs, Journal, Jan 2012.

[18] A. Wendell O. Rodrigues. A Methodology to Develop High Performance Applications on GPGPU Architectures: Application to Simulation of Electrical Machines. Thesis, Université de Lille 1, January 2012. Available from: http://tel.archives-ouvertes.fr/ tel-00670221.

[19] Connie U. Smith and Lloyd G. Williams. Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Addison Wesley Longman Publishing Co., Inc, 2002.

[20] I. Traore, I. Woungang, A.A. El Sayed Ahmed, and M.S. Obaidat. UML-based Performance Modeling of Distributed Software Systems. In Performance Evaluation of Computer and Telecommunication Systems (SPECTS), pages 119 –126, july 2010.

[21] C.M. Woodside. Tutorial Introduction to Layered Modeling of Software Performance. Technical report, Department of Systems and Computer Engineering, Carleton University, Ottawa (Canada), 2002.

Authors

Vincent Aranega received his PhD degree in Computer Science at University of Lille 1 - France in 2011. His research interest includes Model Driver Engineering and more specifically model transformation languages and chaining. He works on the use of model transformation traceability for debugging and profiling purposes.

A. Wendell O. Rodrigues obtained his Ph.D. degree in Computer Science at University of Lille1 - France in 2012. His research interests include parallel architectures and programming, and software engineering, specifically with regards to GPUs and high-level software specification. Rodrigues is currently assistant professor at Federal Institute of Education, Science and Technology of Ceará.

Anne Etien received a Ph.D. degree in Computer Science from the University of Paris 1 Pantheon-Sorbonne, France in 2006. She is now Associate Professor in the University of Lille 1 in France and makes her research at the LIFL. Her area of interest includes model driven engineering. More specifically, she works on various aspects of model transformations, including chaining, evolution, reusability, traceability, genericity.

Frédéric Guyomarch got his PhD in Computer Science in 2000 and is currently assistant profeesor at the University of Lille 1 - France. His research interests focus on high performance computing, from the algorithms for numerical computation to compilation technics to generate optimized code for such algorithms.

Jean-Luc Dekeyser received his PhD degree in computer science from the university of Lille in 1986. He was a fellowship at CERN. After a few years at the Supercomputing Computation Research Institute in Florida State University, he joined in 1988 the University of Lille in France as an assistant professor. He created a research group working on High Performance Computing in the CNRS lab in Lille. He is currently Professor in computer science at University of Lille. His research interests include embedded systems, System on Chip co-design, synthesis and simulation, performance evaluation, high performance computing, model driven engineering, dynamic reconfiguration, Chip-3D.